A General Purpose Automatic Overlapped Tiling Technique in Polyhedral Frameworks

Jie Zhao

INRIA & École Normale Supérieure
45 rue d'Ulm, 75005 Paris, France

ACM Student Research Competition (SRC)
2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)
Vienna, Austria

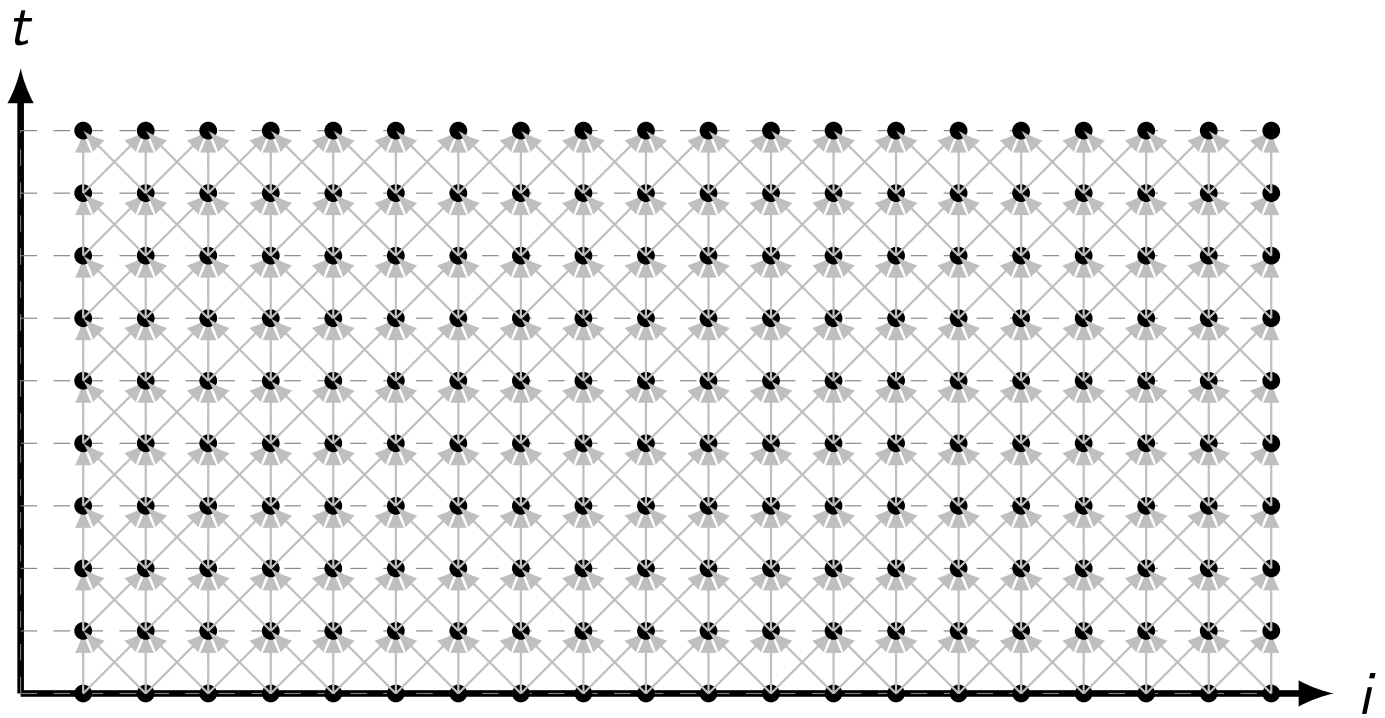
January 26, 2018

Outline

Introduction to Tiling Techniques [IT88]

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

Stencil Code from heat-1d.c

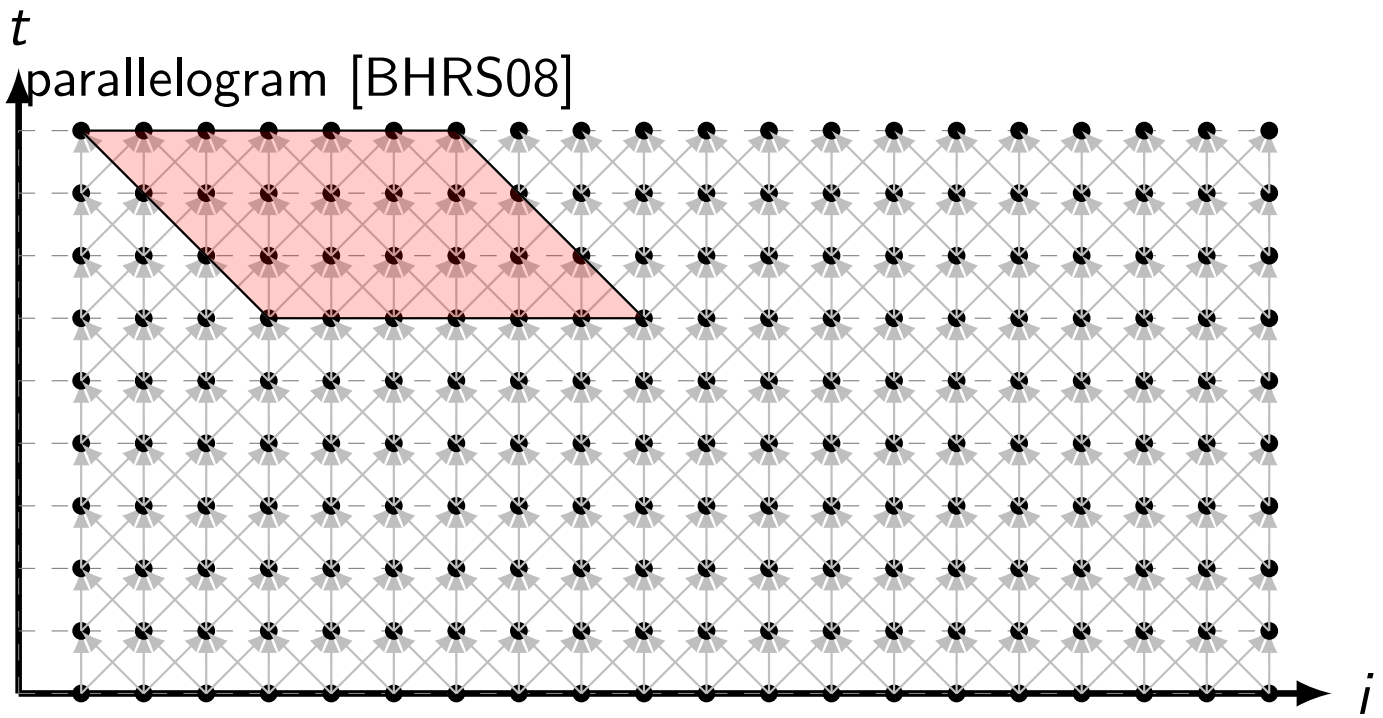


Different tile shapes on the above stencil code's iteration space

Introduction to Tiling Techniques [IT88]

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

Stencil Code from heat-1d.c

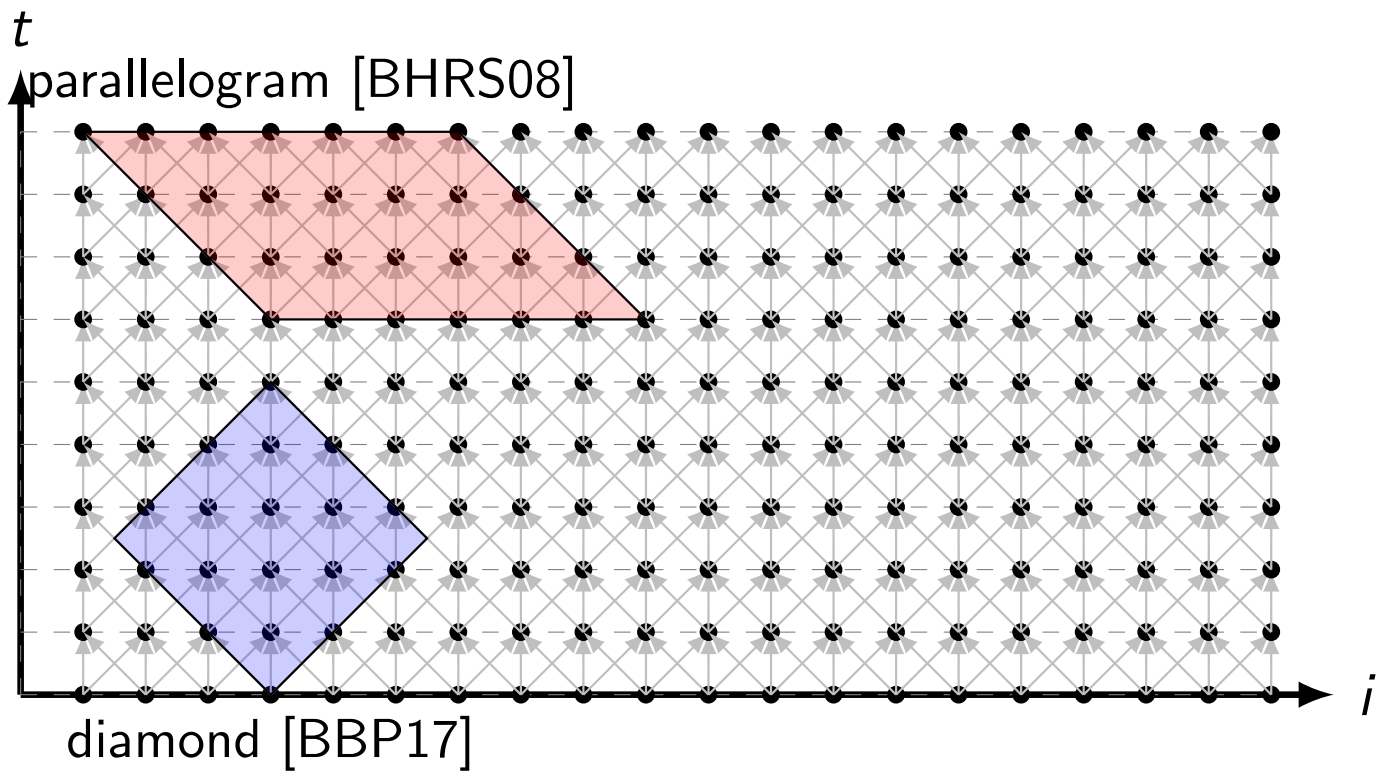


Different tile shapes on the above stencil code's iteration space

Introduction to Tiling Techniques [IT88]

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

Stencil Code from heat-1d.c

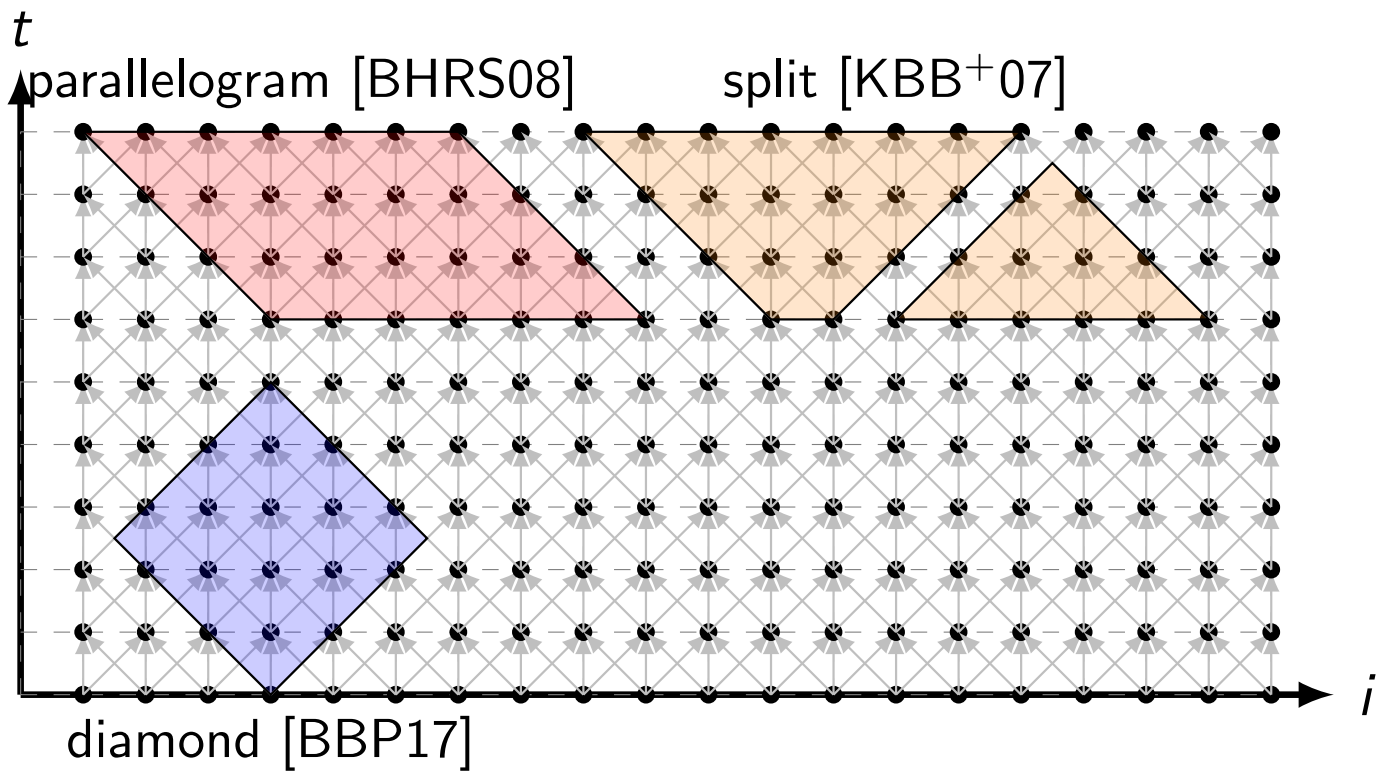


Different tile shapes on the above stencil code's iteration space

Introduction to Tiling Techniques [IT88]

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

Stencil Code from heat-1d.c

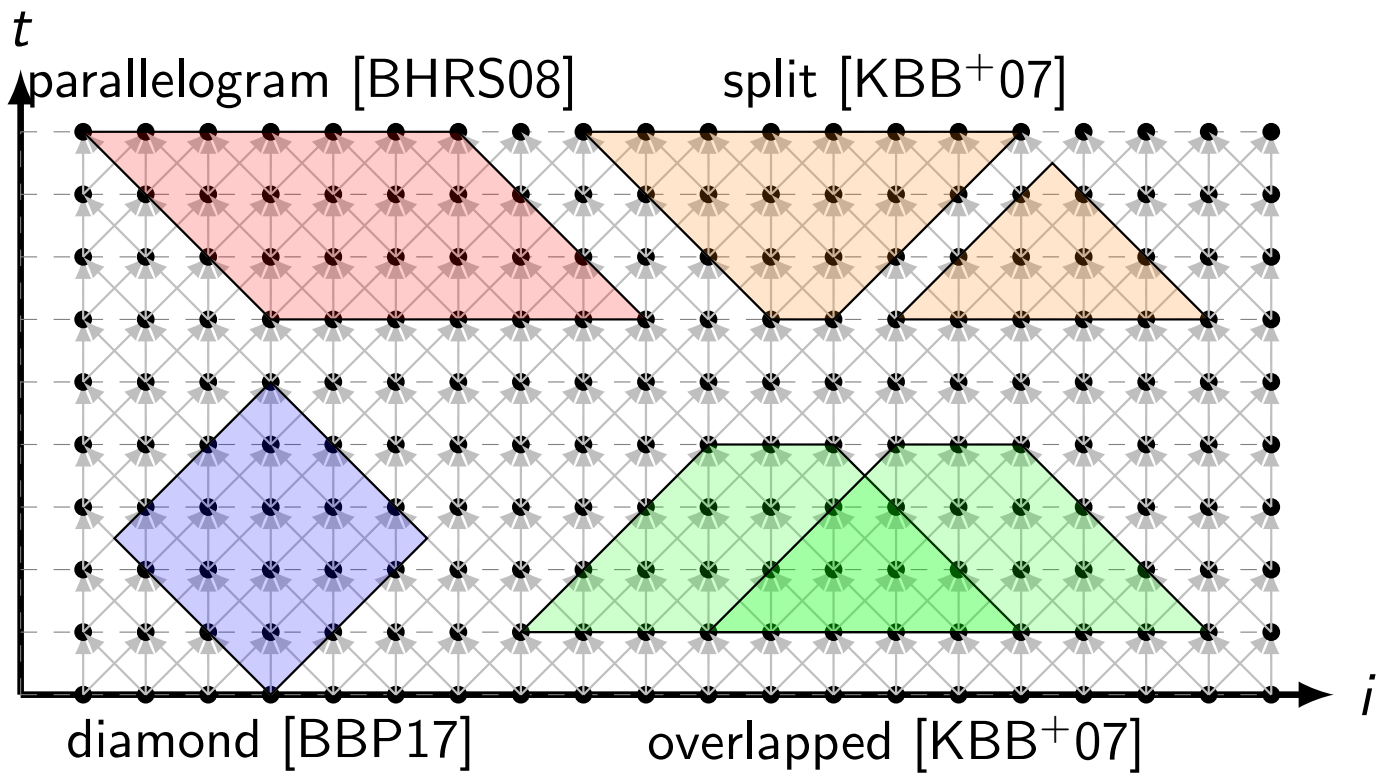


Different tile shapes on the above stencil code's iteration space

Introduction to Tiling Techniques [IT88]

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

Stencil Code from heat-1d.c



Different tile shapes on the above stencil code's iteration space

Comparison Between Different Tiling Techniques

	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

Comparison Between Different Tiling Techniques

	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

- Our goal: implement an effective tiling technique for programs written in general purpose languages:

Comparison Between Different Tiling Techniques

	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—~~parallelogram tiling~~

Comparison Between Different Tiling Techniques

	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—~~parallelogram tiling~~
  - Different target architectures (CPUs and GPUs)—~~diamond tiling~~

Comparison Between Different Tiling Techniques

	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—~~parallelogram tiling~~
  - Different target architectures (CPUs and GPUs)—~~diamond tiling~~
  - Redundancy *v.s.* algorithmic complexity—~~split tiling~~

Comparison Between Different Tiling Techniques

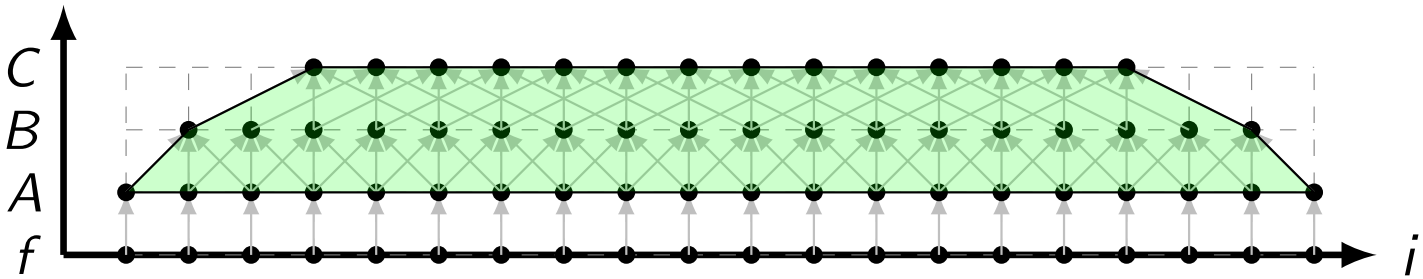
	parallelogram	diamond	split	overlapped
shape complexity	×	✓	✓	✓
parallelism	×	✓	✓	✓
locality	✓	✓	✓	✓
redundancy	×	×	×	✓
algorithmic complexity	×	✓	✓	×

- Our goal: implement an effective tiling technique for programs written in general purpose languages:
  - Concurrent start—~~parallelogram tiling~~
  - Different target architectures (CPUs and GPUs)—~~diamond tiling~~
  - Redundancy *v.s.* algorithmic complexity—~~split tiling~~
  - Overlapped tiling

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

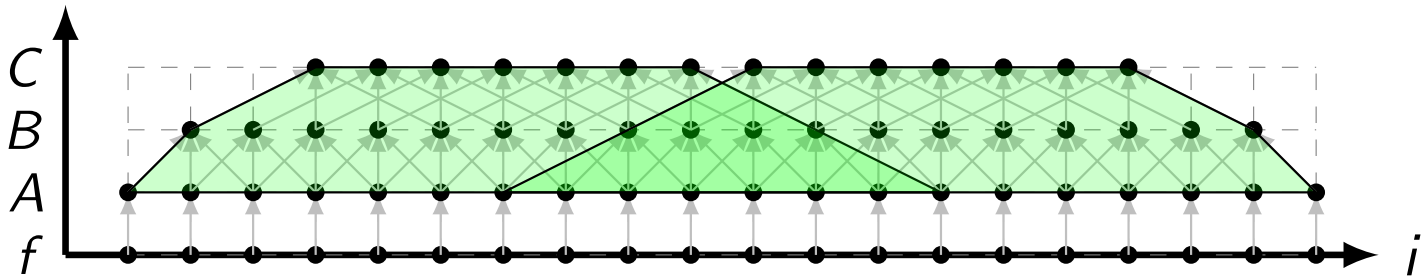


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

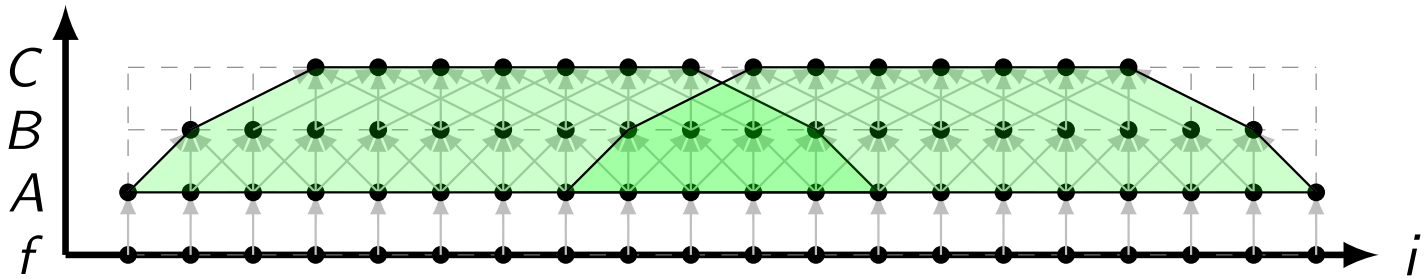


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

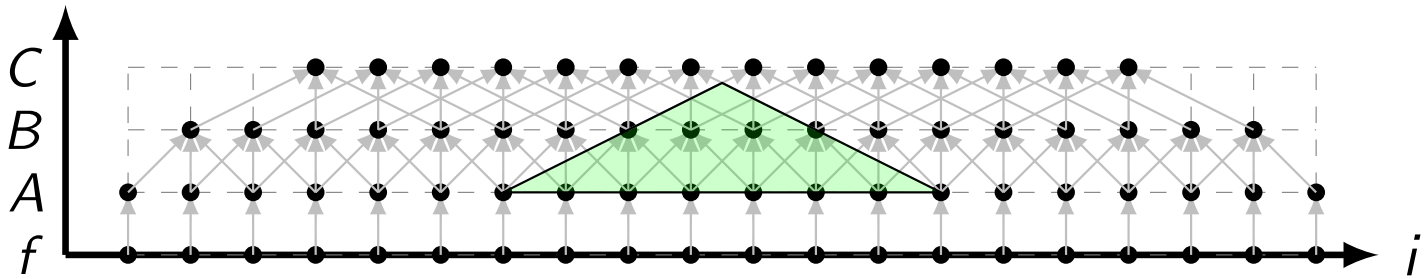


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

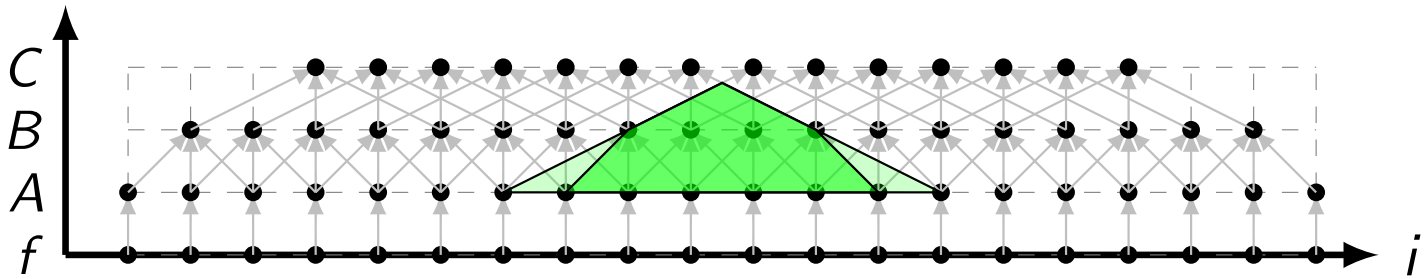


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

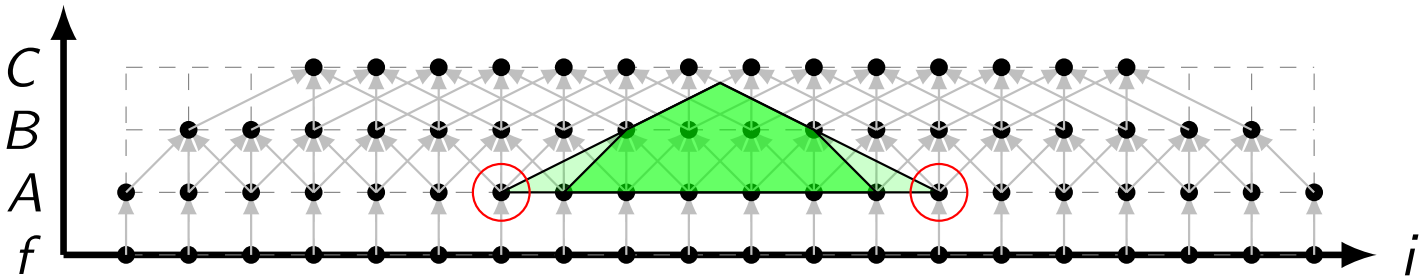


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline

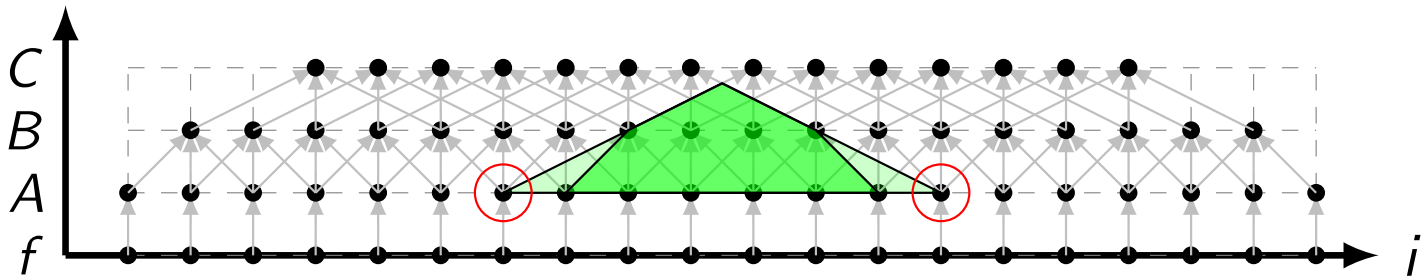


Overlapped tiling on the iteration space

Comparing with PolyMage

```
for (i=1; i<N; i++)
    A[i]=f(i);
for(i=2; i<N-1; i++)
    B[i]=0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i]=0.25*(B[i-2]+2*B[i]+B[i+2]);
```

A simple image processing pipeline



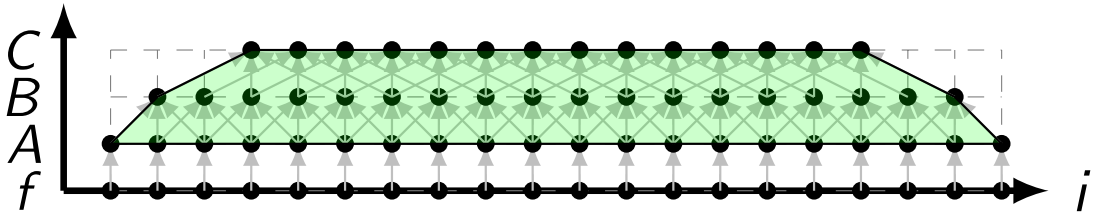
Overlapped tiling on the iteration space

We mitigate redundant computation by shrinking the shadows; better performance than a rescheduling-based technique.

Comparing with PolyMage

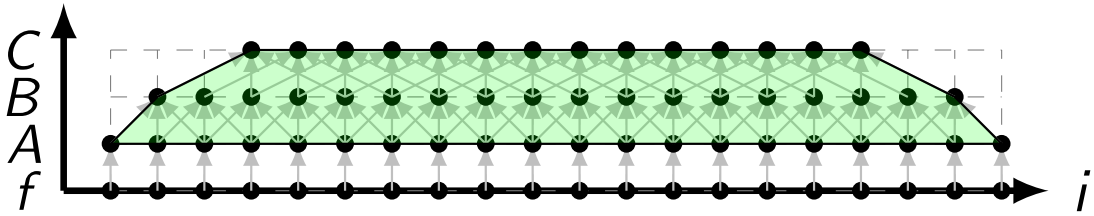
	PolyMage	Our work
Redundancy	More	Less
Implementation	With shifting	With/Without shifting
Applicability	Domain-specific language	General-purpose language
Targets	CPU	CPU & GPU

Overlapped Tiling With Shifting

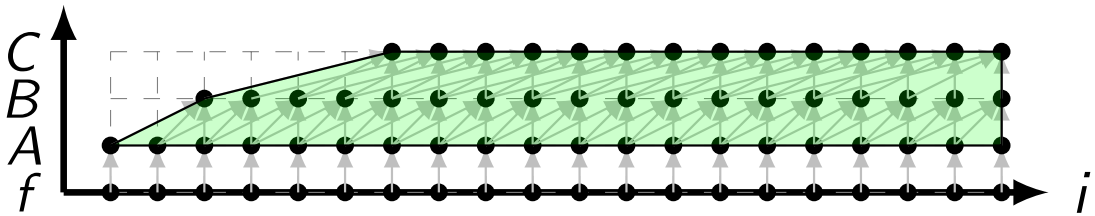


Iteration Space

Overlapped Tiling With Shifting

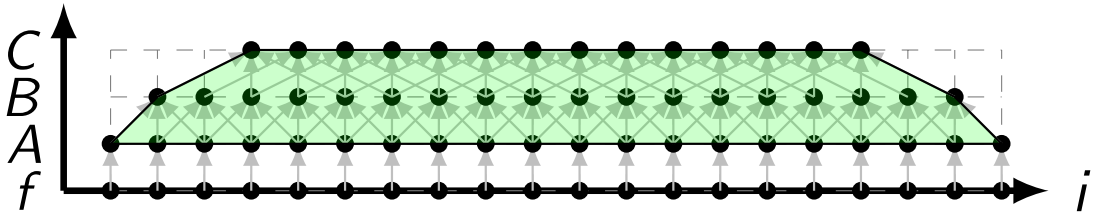


Iteration Space

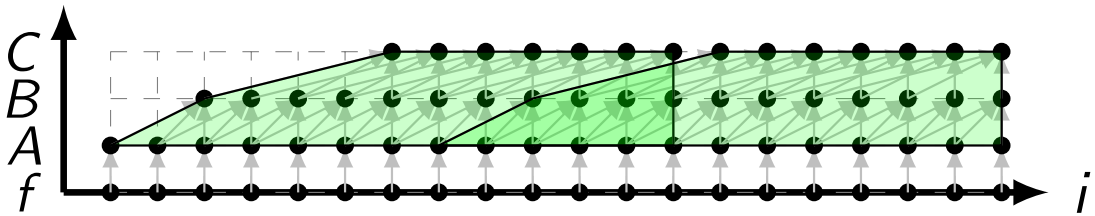


Overlapped tiling with shifting

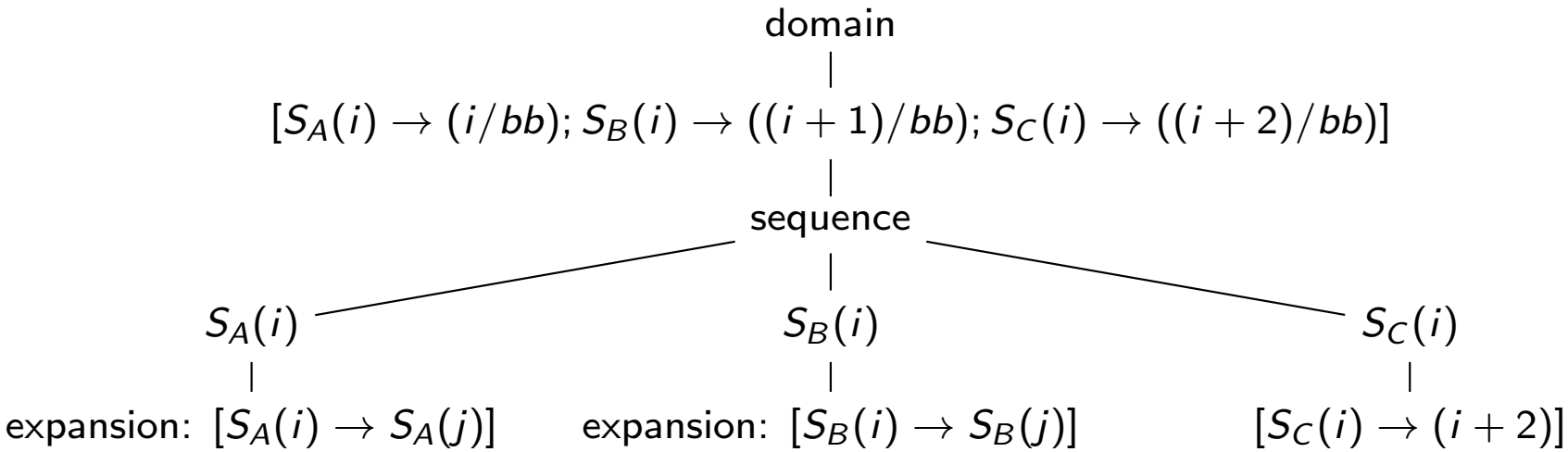
Overlapped Tiling With Shifting



Iteration Space

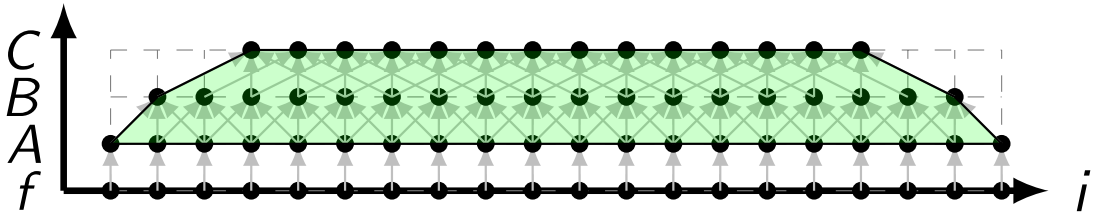


Overlapped tiling with shifting

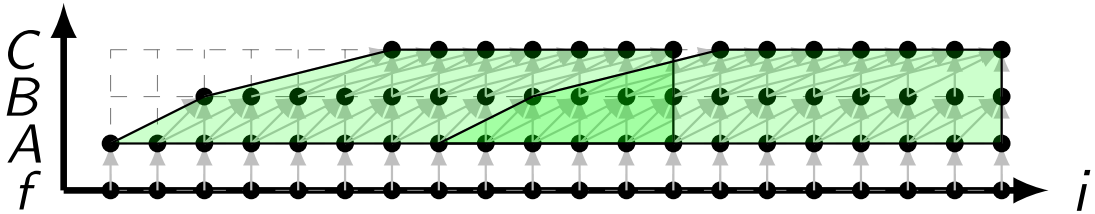


Schedule tree

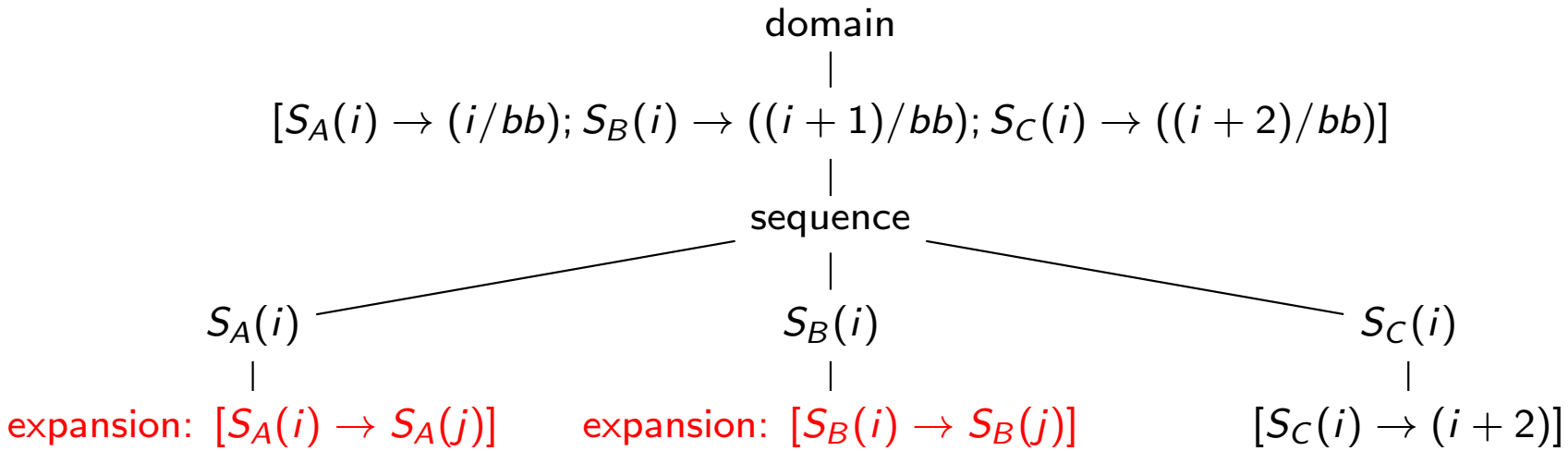
Overlapped Tiling With Shifting



Iteration Space

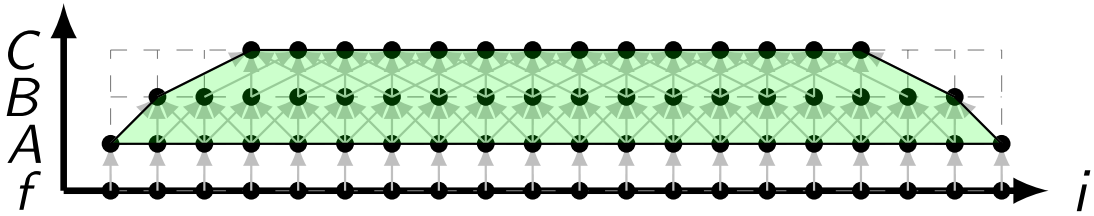


Overlapped tiling with shifting



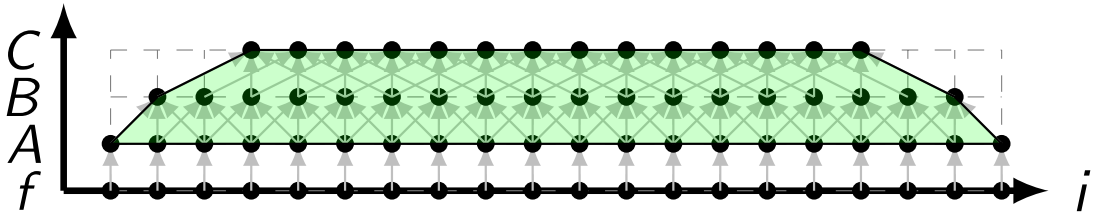
Schedule tree

Overlapped Tiling Without Shifting

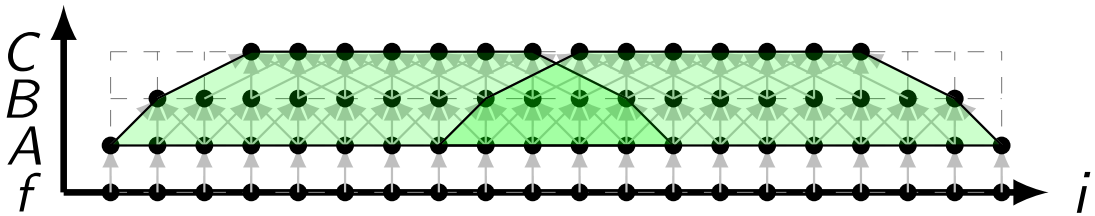


Iteration space

Overlapped Tiling Without Shifting

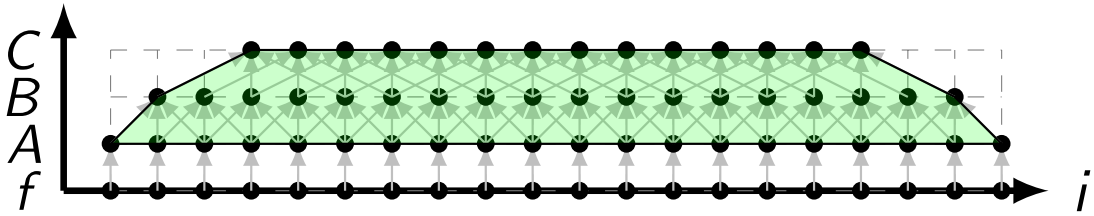


Iteration space

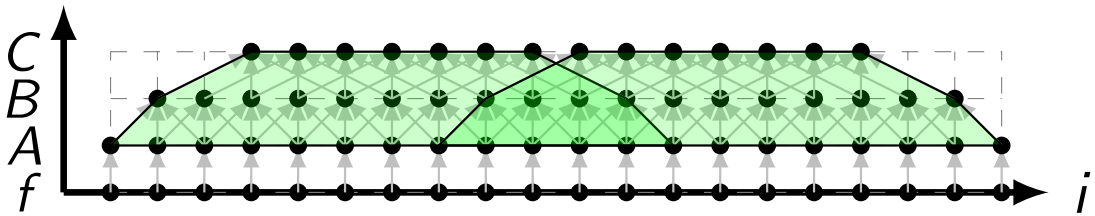


Overlapped tiling without shifting

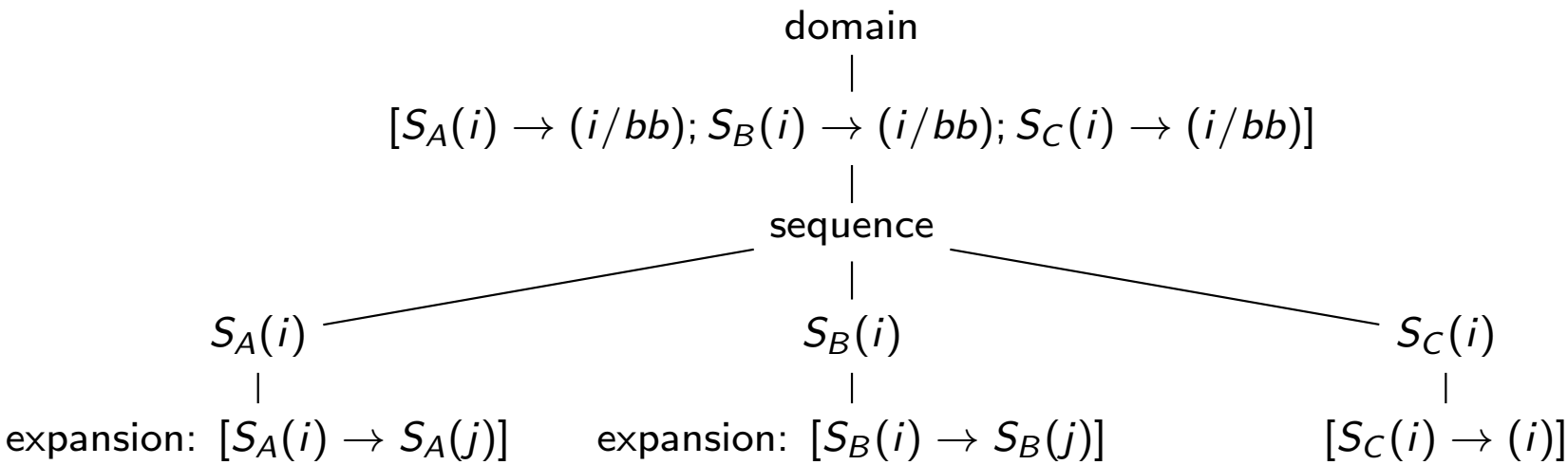
Overlapped Tiling Without Shifting



Iteration space

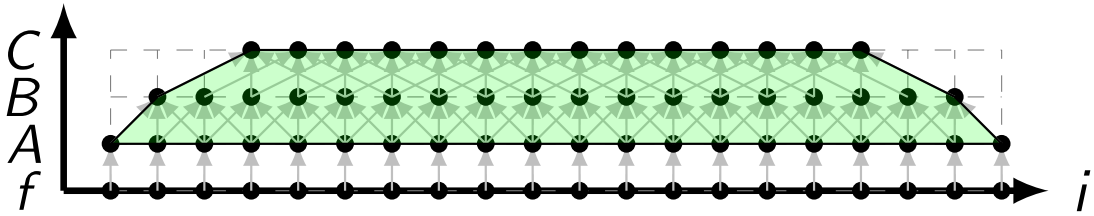


Overlapped tiling without shifting

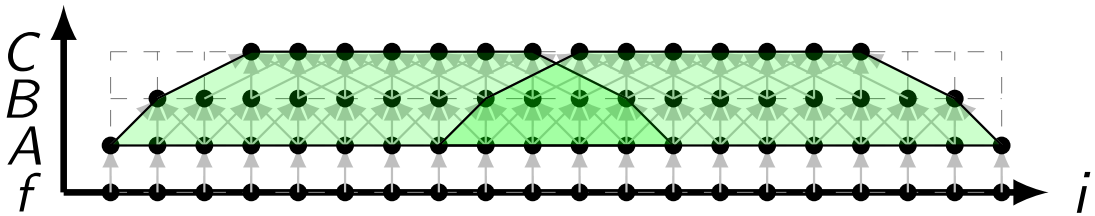


Schedule Tree

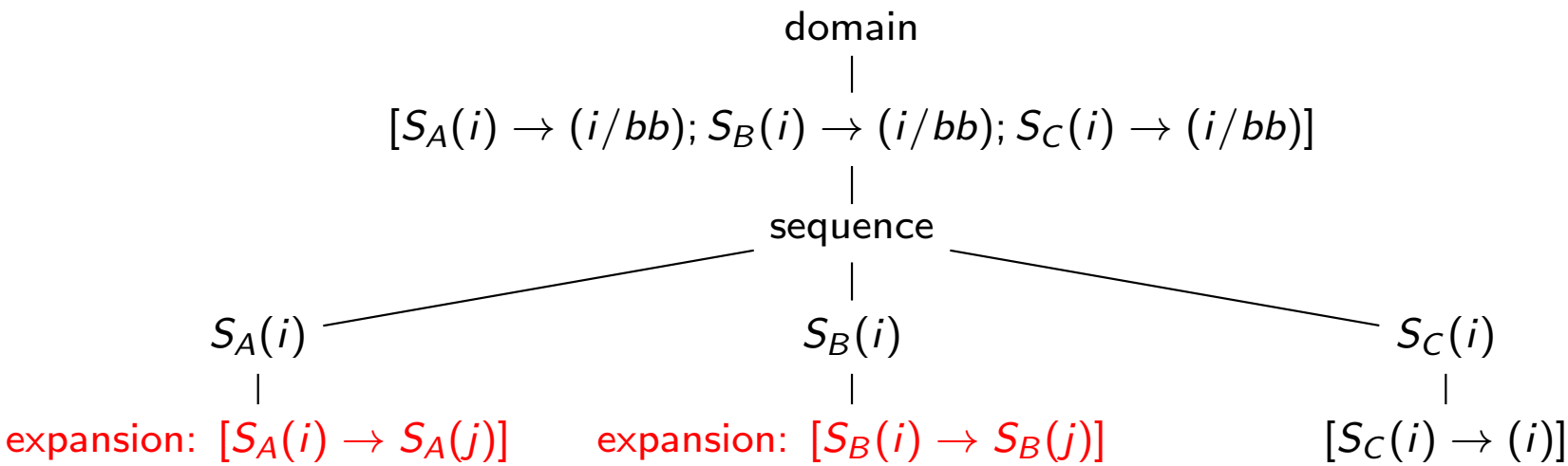
Overlapped Tiling Without Shifting



Iteration space



Overlapped tiling without shifting

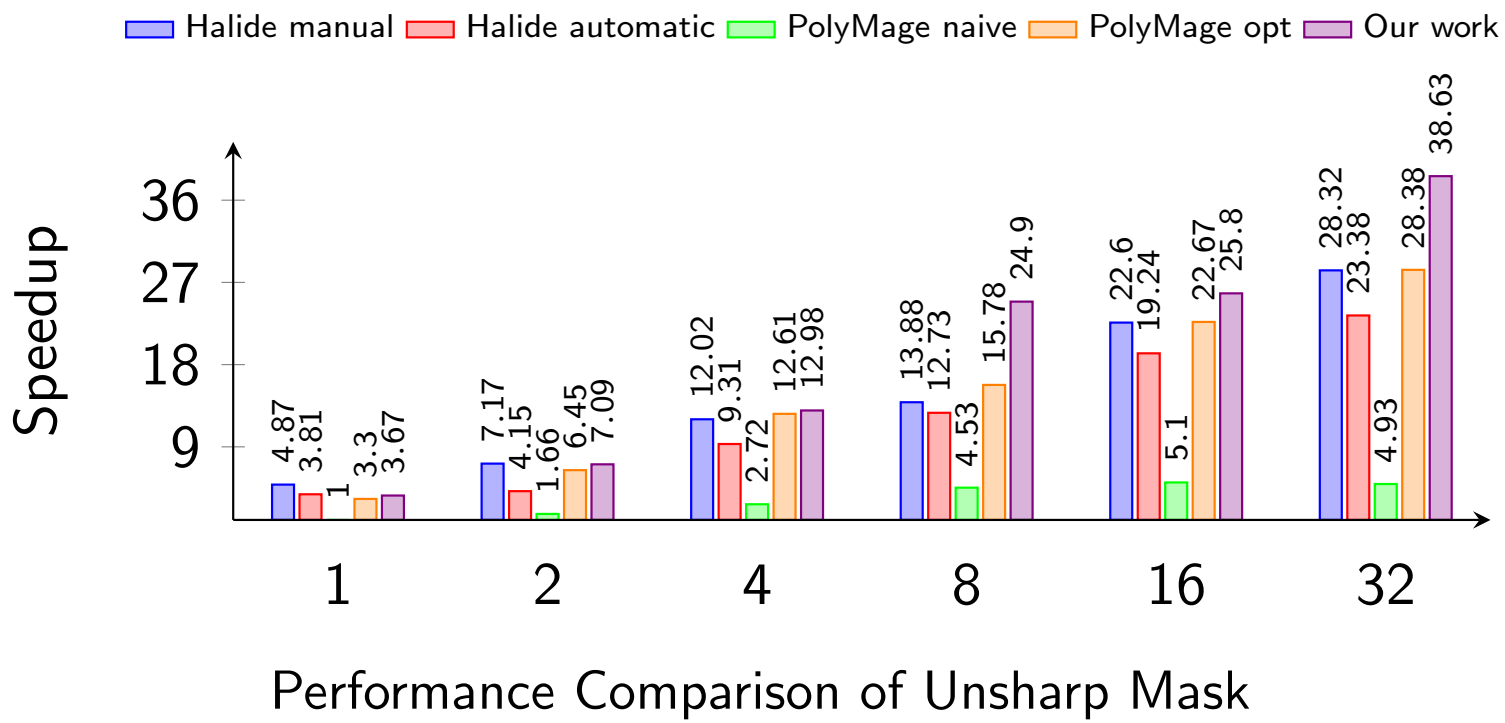


Schedule Tree

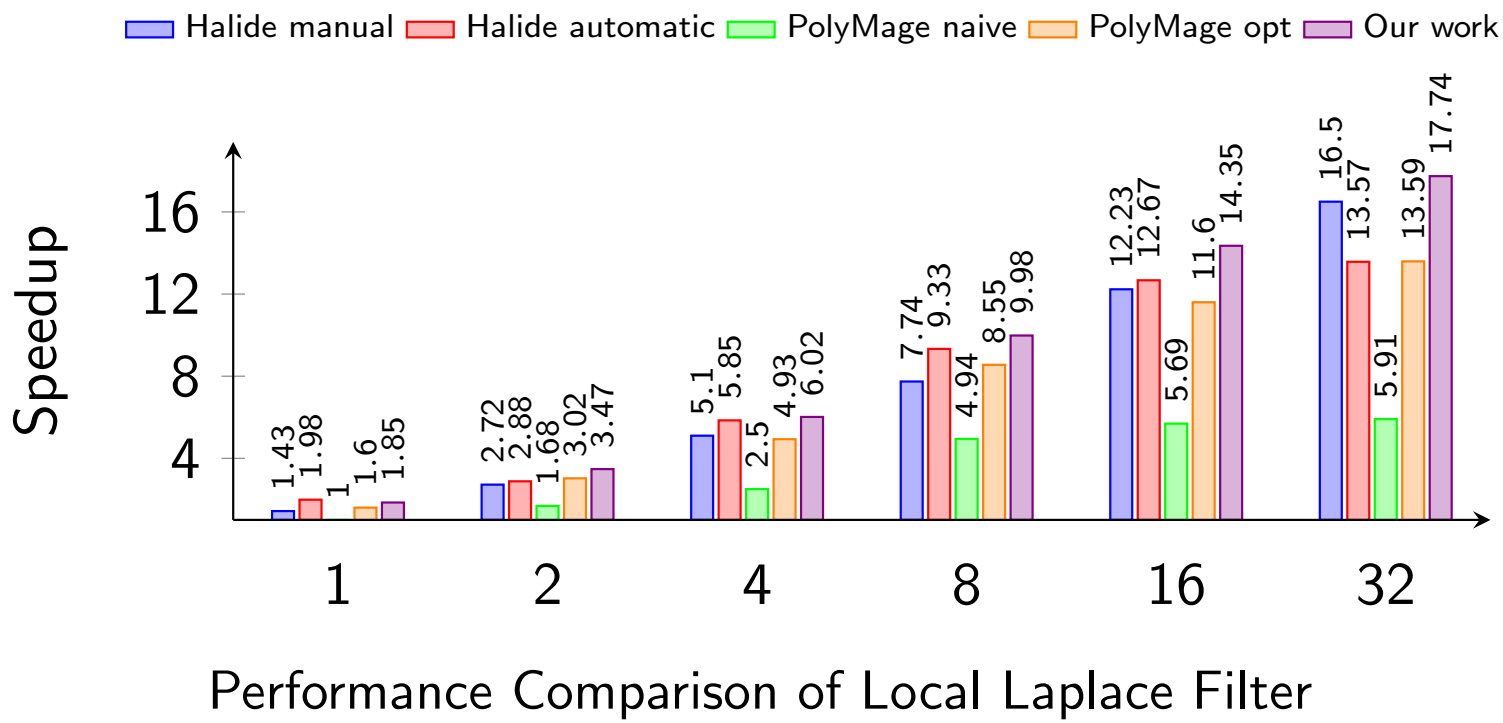
Experimental Setup and Methodology

- Code generator: PPCG (version `ppcg-0.07-26-g236d559`).
- Architecture:
  - CPU: 32-core Intel Xeon(R) E5-2683 v4 @2.10GHz
  - GPU: NVIDIA Quadro K4000
- Compilation:
  - CPU: ICC`18.0` (*-O3 -xHost -qopenmp -ipo*)
  - GPU: NVCC`9.0` (*-O3*)
- Baseline:
  - sequential PolyMage naive code (without tiling) [MVB15]
  - CUDA code generated by PPCG (parallelogram tiling) [VCJC+13]
- Comparison:
  - Halide manual and automatic scheduling [RKBA+13], PolyMage naive and optimized scheduling [MVB15]

Performance Comparison on CPU

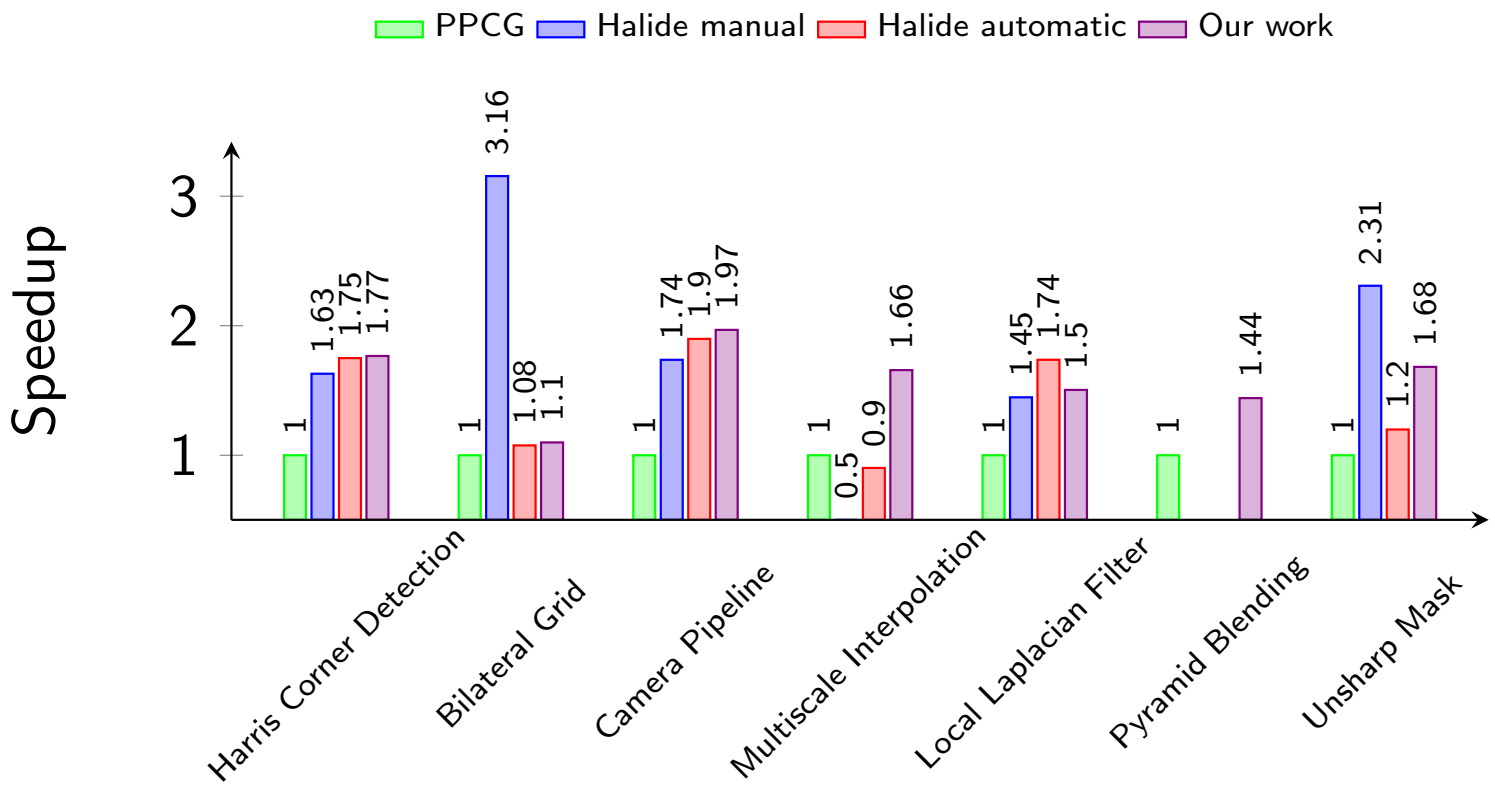


Performance Comparison on CPU



Please refer to our poster for more experimental data on the remaining benchmarks, with a detailed comparison with the state of the art.

Performance Comparison on GPU



Performance Comparison of All Benchmarks on GPU

Our technique is also applicable to iterated stencils. Please check on our poster for the evaluation on both CPU and GPU.

Conclusion and Future Work

- We implemented a general purpose overlapped tiling technique in a polyhedral framework.
- We implemented overlapped tiling technique with/without shifting.
- Our work can generate codes for both CPU and GPU.
- We get better performance due to less redundant computation.
- Future work: finish the experiments and prepare the paper submisson.

References

- Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath.
  Diamond tiling: Tiling techniques to maximize parallelism for stencil computations.
  IEEE Transactions on Parallel and Distributed Systems, 28(5):1285–1298, October 2017.

- Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan.
  A practical automatic polyhedral parallelizer and locality optimizer.
  In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

- F. Irigoin and R. Triolet.
  Supernode partitioning.
  In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

- Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan.
  Effective automatic parallelization of stencil computations.
  In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

- Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula.
  Polymage: Automatic optimization for image processing pipelines.
  In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

- Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe.
  Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
  In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

- Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor.
  Polyhedral parallel code generation for cuda.
  ACM Trans. Archit. Code Optim., 9(4):54:1–54:23, January 2013.